Rollback and Recovery Strategies for Computer Programs

K. M. CHANDY AND C. V. RAMAMOORTHY, MEMBER, IEEE

Abstract—Reliability is an important aspect of any system. On-line
diagnosis, parity check coding, triple modular redundancy, and other
methods have been used to improve the reliability of computing systems. In
this paper another aspect of reliable computing systems is explored.
The problem is that of recovering error-free information when an error is
detected at some stage in the processing of a program.

If an error or fault is detected while a program is being processed and
if it cannot be corrected immediately, it may be necessary to run the
entire program again. The time spent in resuming the program may be
substantial and in some real time applications critical. Recovery time
can be reduced by saving states of the program (all the information stored
in registers, primary and secondary storage, etc.) at intervals, as the process-
ing continues. If an error is detected the program is restarted from its most
recently saved state. However, a price is paid in saving a state in the
form of time spent storing all the relevant information in secondary stor-
age. Hence it is expensive to save the state of the program too often. Not
saving any state of the program may cause an unacceptably large recovery
time. The problem that we solve is the following. Determine the optimum
points at which the state of the program should be stored to recover after
any malfunction.

Index Terms—Computer programs, fault tolerance, graph models, re-
cover, reliability, rollback.

INTRODUCTION

RAPID and smooth restoration of a computing system
after an error or malfunction is always a major
design and operational goal. Hardware failures can
be detected and corrected by suitable diagnostic and main-
tainability procedures. Software design errors could be hard
to detect but once detected corrective procedures would be
easy to implement. The majority of operational failures
occur in the hardware processors, memory, and I/O. On-
line diagnosis and use of error checking codes have been
effective in reducing the effects of these hardware malfunc-
tions. After the malfunction is corrected, the problem arises
as to where to restart the program. It may not always be
feasible to run the entire set of programs again from the
start, either due to time limitations or since the required
data has been modified. A better strategy would be to have
a number of rollback points (or check points) within the
program at which certain program and processor status
information could be saved. If a fault or malfunction is
detected, the program is rolled back to a previous check point
where the system is known or proven to be in good oper-
tional condition.

Various strategies are used to reduce the impact of inter-
ruptions or malfunctions both to the system and to the
users. Operating System 360 as used in Model 65 is equipped
with a set of programs called the recovery management sup-
port which embodies a number of methods. The recovery
methods depend upon the nature of the malfunction. In the
I/O area, rereading of input data with parity errors is com-
mon. If error subsists even after repeated retries the system
could consider reconstruction of damaged data (error correc-
tion) if possible. In the case of the processor errors, the
instruction may be retried if feasible (i.e., if its operands
were not modified by the instruction). The most important
technique is to provide check points in all programs so that
programs could be rolled back to a previous state and com-
putation resumed.

If an error is detected while a program is being processed
and if the error cannot be corrected immediately, it may be
necessary to run the entire program again. The time lost in
running the program again may be substantial and in some
real time applications (notably aerospace and process con-
tral) critical.

At any stage in the processing of a program certain in-
formation is required by the program for computation to
proceed successfully. A state at any stage in the processing
of a program, will be defined as the information (variables,
data, programs) which may be subsequently used by the
program. Saving the state of a program is the process of
making a copy of the state in secondary storage. Clearly, the
length of time spent in saving a state is proportional to the
amount of information that has to be copied. Loading a
saved state is the process of setting all the registers, primary
and secondary storage etc. to the values stored in them when
the state was saved. Recovery time can be reduced by saving
states of the program at intervals, as the program gets pro-
cessed; if an error is detected the program is restarted from
its most recently saved state. If the states of the program are
saved too frequently, an unnecessarily large amount of time
may be spent in saving states. If the states of the program are
saved too infrequently an unacceptably large recovery time
may result. The resolution of the tradeoff is the subject of
our discussion.

Only transient malfunctions are treated in this paper. It
is clear that permanent malfunctions cannot be treated by
rollback alone, since, if a permanent malfunction does occur
and is detected, and the system starts recomputing from a
rollback point, the very same permanent malfunction will be
detected again. On the other hand, rollback is a very useful
tool for handling permanent malfunctions, when it is used
with some other fault-tolerant technique that effectively
switches off the malfunctioning device. This is discussed in
greater detail in what follows.

We will now discuss some areas where rollback can be
profitably combined with other fault-tolerant techniques.
The earliest attempts at obtaining ultrareliable systems attempted to achieve reliability through redundancy [4]. Triple modulo redundancy (TMR) and other methods of fault-tolerant computing using several identical computing units, operating in parallel on the same data with a vote taker (see Fig. 1), have been discussed in great detail [5]–[7]. A slightly different system, using TMR and standby spares that are switched in when needed, has been described by Mathur and Avižienis [8]. This system is called the hybrid system. The operation of the hybrid system, in brief, is as follows. Three identical computing units are operated in parallel and a vote taker compares the outputs of each unit with the others (see Fig. 2). If the output of one unit does not tally with the other two, it is switched out and a standby spare is powered on to take its place. However, after the standby unit is powered on, the registers, memory, and program status word must be loaded with the appropriate information before processing can continue. One way of doing this is to have rollback points; the three units, including the standby unit are loaded with the information saved at the last rollback point, and processing continues from there.

Any system that uses spares is confronted with the problem of loading the powered on spare. One method for solving this problem is to use rollback. Rollback at periodic intervals was used in the SABRE 7090 System and in the IBM 9020 System used by the FAA.

We will make the assumption that if an error occurs while a task is being processed, then the error is diagnosed before the task is completed. Suppose an error occurs while a task is being processed and suppose it is not diagnosed; if there is a rollback point immediately after the task is complete, then the information that is saved at the rollback point will be faulty. Subsequently, if the error is diagnosed, this faulty information will be loaded, and the computer will continue processing from the rollback point; eventually the same error will be diagnosed again.

If the same error is detected after rolling back, the system should conclude that there is either a permanent malfunction, or that an undiagnosed error occurred before the last rollback point. The program may be rerun from the very beginning and if the same error is detected again, one may reasonably conclude that a permanent malfunction has occurred, and a reconfiguration made to switch off the faulty unit.

Rollback can be used in two quite different ways. In some systems, the programmer preanalyzes his program and specifies where rollback points are to be inserted. He may decide where to insert rollback points in either an intuitive manner or by making estimates about relevant parameters in his program (such as the maximum time that may be required to process a given task in the program), and by using a mathematical model to aid him in the decision making. In other systems rollback points are inserted at periodic intervals [14], irrespective of the particular programs being run. We are concerned with the former case, where rollback points are tailor-made for a particular program.

The amount of information that has to be saved so as to be able to restart a program at that point may vary widely from one point in the program to the next. We assume that there is a sufficient amount of secondary storage to store the state of a program at any time. The secondary storage used may be a large core storage unit [9], drum, disk, or even magnetic tape. The “cost” associated with a rollback point is the time taken to save the state of the system at that point; the time clearly depends on the amount of information that has to be stored and on the type of memory used to store the saved state. These factors are included in the mathematical model described later.

This paper uses a graph model to describe a program. Graph models have been dealt with extensively in the literature [10]–[13]. Programmers have traditionally used flow charts (which are graphs of a kind) as aids in programming. In this paper, we assume that a programmer can analyze his program (or flow chart), and represent it as a sequence of tasks. A task may be an instruction, or several instructions including conditional branches. In our paper, we will generally make a coarse partition of the program into tasks, i.e., each task will consist of several instructions and will involve a substantial amount of processing time; the range implied

---

Fig. 1. Triple modulo redundancy.

Fig. 2. Hybrid system.
The sequence of tasks processed may change from one run of the program to the next due to conditional branch statements. However, we shall assume that no task is repeated; if a task is iterated in a program, each iteration of the task may be considered a distinct task, or the iterations may be coalesced in the manner shown in what follows. Consider the flow chart in Fig. 3, where a task is iterated \( n \) times. The iterations may be coalesced into one task, or into a sequence of one or more tasks; each of the tasks in the program graph may correspond to several iterations of the task in the flow chart.

The algorithm makes use of estimates made by the programmer on the maximum amount of processing time required by a task. Admittedly, it is impossible to design an algorithm, which, given any program, determines the maximum time that may be required to process each task in the program. However, it is possible for a programmer to obtain estimates of worst case bounds for his particular program. Indeed, in many computer installations, programmers have to submit estimates of the maximum time required to process their jobs. It is important to note that in installations where a programmer is allowed to specify rollback points, he must make estimates of this sort, and then make intuitive decisions based on these estimates. Our objective is to clarify, quantify, and formalize his decision-making process. The accuracy of the decisions (intuitive or formalized) clearly depends on the accuracy of the estimates.

Obtaining a program graph from a program is not inexpensive. The programmer must analyze his flow charts and make estimates of several parameters. Many (probably most) programs are short enough so that no rollback points at all are required. In many other cases, the advantage gained in having tailor-made rollback points is not worth the time spent by a programmer in obtaining a program graph; in these cases rollback points at fixed intervals are sufficient. However, there are some cases, where the costs of slow error recovery are high, where the system runs a comparatively small set of programs over and over again, and where the advantage of tailor-made rollbacks outweighs the time spent by the programmer in constructing the program graph. We are concerned with cases of the latter type.

The decision to insert rollback points clearly depends on the importance of speedy error recovery, i.e., the penalty incurred if a program does not run to completion in a prescribed amount of time. In some real time applications it is critical that a program run to completion in some given amount of time, whereas, in most commercial applications the loss incurred if an error occurs is just the computer time wasted.

A programmer has to analyze his flow chart and represent it as a sequence of tasks (program graph) only once. Hence, the greater the number of times a program is run, the more the benefit of tailor-made rollback points. So, the decision to have tailor-made rollbacks clearly depends on the expected number of times the program will be run.

Programs with short processing times do not need rollback points at all. Thus, a program that is worth analyzing for tailor-made rollbacks must have three characteristics. 1) The program must require a substantial amount of processing time; 2) the application of the program must be such that quick error recovery is crucial; and 3) the same program must be run a large number of times.

**Problem Formulation**

A program will be represented by an undirected graph where vertex \( i \) corresponds to task \( i \) and edge \((i, j)\) exists if and only if task \( i \) is followed by task \( j \) with nonzero probability.

Associated with vertex \( i \) of the graph is a real number \( t_i \) which is the maximum (or expected) time between the start and the completion of task \( i \).

Associated with each edge \((i, j)\) of the graph are two real numbers: \( S_{ij} \) and \( L_{ij} \). The state of the program soon after task \( i \) is completed and before task \( j \) is started (if task \( j \) is processed next) is described by the program status word, register contents, primary and secondary storage contents, and so forth. The time taken to save (make a copy of) the state of the system at this stage in the program in secondary storage is \( S_{ij} \). We shall refer to \( S_{ij} \) as the save time. The time taken to load the state of the system from secondary storage to primary storage is \( L_{ij} \).

At each edge \((i, j)\) we may choose to insert, or to not insert a rollback point. If a rollback point is inserted at edge \((i, j)\) then after task \( i \) is completed, and if task \( j \) is to be processed
next, the state of the system is saved in secondary storage before task \( j \) is started and any prior state that has been saved earlier is erased. Subsequently, if a transient error occurs, the program is restarted at the most recently saved state.

We define the **recovery time** \( r \) at any point \( P \) in the program to be the time taken to load the most recently saved state and to rerun the program from this state to \( P \). If an error is detected at point \( P \), the recovery time \( r \) is the time "lost" due to the error. The question that we wish to answer is the following: Where should rollback points be inserted?

There are several formulations of the problem. Three of the models are discussed in what follows. In all models we assume that if an error occurs while task \( i \) is being processed then the error is detected before task \( i \) is completed.

**Worst-Case Design**

**Data:** With every task \( i \) we associate a real number \( t_i \), where \( t_i \) is the maximum processing time that will be required by task \( i \). \( L_{ij} \) and \( S_{ij} \) are the maximum load and save times if a rollback point is inserted on edge \((i,j)\). We are also given \( M \), the maximum recovery time.

**Constraints:** Insert rollback points so that at every point in the program the maximum possible recovery time does not exceed \( M \).

**Objective Function:** Minimize the maximum time (i.e., for the worst case) that may be spent in saving states of the system in secondary storage.

**Minimal Expected Save-Time Design**

**Data:** Associated with task \( i \) is a real number \( t_i \), where \( t_i \) is the expected time required to process task \( i \). A real number \( p_{ij} \) is associated with each edge \((i,j)\) of the graph, where \( p_{ij} \) is the probability that task \( i \) will be immediately followed by task \( j \). \( L_{ij} \) and \( S_{ij} \) are the expected load and save times if a rollback point is placed on \((i,j)\). We are also given \( M \), the maximum expected recovery time.

**Constraints:** The expected recovery time at any point in the program is not to exceed \( M \).

**Objective Function:** Minimize the expected time spent in saving states of the system in secondary storage.

**Minimal Expected Run-Time Design**

**Data:** \( p_{ij} \) is a real number associated with each edge \((i,j)\) where \( p_{ij} \) is the probability that task \( i \) is immediately followed by task \( j \). We associate a probability \( Q_i \) with task \( i \) where \( Q_i \) is the probability that a transient error will occur while task \( i \) is being processed. Given that a transient error does occur while task \( i \) is being processed, let \( y_i \) be the time between the initiation of task \( i \) and the occurrence of the error. \( y_i \) is a random variable; we assume that the probability distribution function for \( y_i \) is known. Given that a transient error will not occur while task \( i \) is being processed, let \( t_i \) be the time required to process task \( i \). \( t_i \) is a random variable; we assume that the probability distribution function for \( t_i \) is known. We shall assume that the save and load times, \( S_{ij} \) and \( L_{ij} \) are constants.

All events are assumed to be independent.

**Constraints:** None.

**Objective Function:** Minimize the expected run time of the program.

**A COMPARISON OF THE DIFFERENT FORMULATIONS**

A programmer can generally provide an estimate of the maximum time that a task will require to get processed; he usually finds it more difficult to estimate the probability distribution function for the processing time required by any given task. For this reason, it is not possible to use the minimal expected run-time design unless a substantial amount of measurement can be carried out on the program so as to estimate the distribution functions for processing times of all the tasks.

The estimation of the probability that the program will branch in any particular direction is also difficult, without substantial measurement. For these reasons the worst-case design is the most pragmatic method of designing rollback points when there are few statistics available.

The best model to use depends on the function of the program as well as on the information available.

The worst case design and the constrained expected recovery-time design are discussed in this paper. The minimal expected run-time design is the topic of a subsequent paper.

We shall first consider worst-case design.

**Implementation**

It is not possible to predict precisely how much processing time a given task will require. It therefore seems desirable to make insertion of rollback points a dynamic procedure; on some runs of a program it may be preferable to have a rollback point on a particular edge while on other runs of the same program (with different data) it may be preferable not to have a rollback point on that edge. However, the procedure for making the decision on inserting rollback points should be simple so that the decision can be made in real time with little overhead. The method suggested here fulfills these requirements.

We interrogate the recovery time \( r \) after each task completion and use it as a basis for making the decision on placing rollback points. \( r \) can be determined readily. Let \( D \) be the clock time at the end of the last rollback, \( L \) the time required to load the system at the last rollback point, \( E = D - L \), and "clock" the current clock time. Then \( r = \text{clock} - E \).

Suppose that at some point in the program the task just completed and the task to be processed next are \( i \) and \( j \), respectively. Let \( r \) be the recovery time at this point. We show that the optimal decision is to insert a rollback point if \( r > B_{ij} \) and not to insert a rollback point if \( r \leq B_{ij} \), where \( B_{ij} \) is a constant. The set of \( B_{ij} \) are computed before the program is run. When task \( i \) is completed and if task \( j \) is to be processed next, \( r \) is compared with \( B_{ij} \) and a rollback point is inserted if \( r > B_{ij} \). If a rollback point is inserted, then \( E \) is updated. Task \( j \) is then processed. A block diagram is presented in Fig. 4. In general \( r \) will vary from one run of the program to the next since the time required to execute a
task will depend on the input parameters. Hence the insertion of rollback points will also vary from run to run of the program, since the decision to insert rollback points is based on the value of \( r \).

**Definitions**

If there exists a path from vertex \( i \) to vertex \( j \) then vertex \( j \) is said to be a successor of vertex \( i \). A vertex with no successors is called an exit vertex.

For each vertex \( i \) in the graph we determine a function \( f_i(r) \), for all possible values of recovery time \( r \), where \( f_i(r) \) is the minimum time spent in saving states of the system after task \( i \) is completed and before the completion of the program, in the worst case. Since we are using worst-case design, when predicting the amount of time required by a task \( i \) we always assume the worst, i.e., that task \( i \) will require the maximum processing time \( t_i \). Similarly, in predicting the branch that a program will take, we assume the worst, i.e., that a program will branch such that the largest amount of time will be spent in saving states of the system.

For each edge \((i, j)\) in the graph we determine functions \( g_{ij}(r) \) and \( x_{ij}(r) \): \( g_{ij}(r) \) is the minimum time spent in saving states of the system after task \( i \) is completed and before the completion of the program, in the worst possible case, if task \( i \) is followed by task \( j \).

\( x_{ij}(r) \) is the optimal decision variable; \( x_{ij}(r) = 1 \) if a rollback point is to be inserted on edge \((i, j)\) and \( x_{ij}(r) = 0 \) otherwise.

**The Algorithm:** We assume \( f_i(r) = \infty \) for all \( i \), and \( r > M \).

**Initialization (0th Step):** Define \( f_i(0) = 0 \), \( r < M \), if \( i \) is an exit vertex. Label all exit vertices. (Vertex \( i \) is labeled to show that the function \( f_i(\cdot) \) has been determined for it.)

**kth Step, \( k = 1, 2, 3, \ldots \):** Determine if there exists any vertex that has all of its successors labeled. If no such vertex exists, stop, the algorithm terminates. If such a vertex exists let it be vertex \( i \).

For all edges \((i, j)\) compute \( g_{ij}(r) \) and \( x_{ij}(r) \) from

\[
g_{ij}(r) = S_{ij} + f_j(L_{ij} + t_j), \quad \text{if } r + t_j > M
\]

\[
g_{ij}(r) = \min \{ f_j(r + t_j), S_{ij} + f_j(L_{ij} + t_j) \}, \quad \text{if } r + t_j \leq M
\]

\[
x_{ij}(r) = 0, \quad \text{if } g_{ij}(r) = f_j(r + t_j)
\]

\[
x_{ij}(r) = 1, \quad \text{otherwise.}
\]

**Theorem 1:** The algorithm determines the optimal decision rules \( x_{ij}(r) \) for each edge \((i, j)\).

**Proof:** We shall show by induction on the \( k \)-th step that if vertex \( i \) is labeled on the \( k \)-th step, then \( x_{ij}(r), g_{ij}(r) \) computed by (1), (2), and (3a) satisfy the definitions given earlier.

**Basis:** If vertex \( i \) is labeled on the first step, then (1), (2), and (3a) reduce to

\[
x_{ij}(r) = 0, \quad \text{if } r + t_j \leq M
\]

\[
x_{ij}(r) = 1, \quad \text{if } r + t_j > M
\]

\[
f_i(r) = g_{ij}(r) = 0, \quad \text{if } r + t_j \leq M
\]

\[
f_i(r) = S_{ij}, \quad \text{if } r + t_j > M
\]

A rollback point must be inserted on edge \((i, j)\) if \( r + t_j > M \), for otherwise the recovery time after task \( j \) is completed may
CHANDY AND RAMAMOORTHY: ROLLBACK AND RECOVERY STRATEGIES

exceed $M$. A rollback point need not be inserted on edge $(i, j)$ if $r+t_j > M$, and if vertex $j$ is an exit vertex. Hence, the theorem is trivially true for $k=1$.

Induction Step: Assume the induction hypothesis to be true for $k=1, 2, \ldots , t-1$. We shall prove it to be true for $k=t$.

If a rollback point is inserted on edge $(i, j)$ the maximum recovery time after task $j$ is $L_{ij} + t_j$. If the recovery time after task $j$ is completed is $L_{ij} + t_j$, then the minimum time spent in saving states of the system after task $j$, in the worst possible case, is $f(L_{ij} + t_j)$ by the induction hypothesis. $S_{ij}$ units of time are spent in saving the state of the system between tasks $i$ and $j$. Hence the minimum time spent in saving states of the system after task $i$ is completed and if task $j$ is processed next, in the worst case is

$$S_{ij} + f_j(L_{ij} + t_j).$$

If a rollback point is not inserted on edge $(i, j)$ the maximum recovery time immediately after task $j$ is $r + t_j$. Hence, in this case the minimum time spent in saving states of the system after task $i$, and if task $j$ is processed next, in the worst case, is

$$f_j(r + t_j).$$

If $r + t_j > M$, a rollback point must be inserted on edge $(i, j)$ if the recovery time after task $j$ is completed is not to exceed $M$.

If $r + t_j \leq M$, we have the option of not inserting a rollback point ($x_{ij}(r)=0$) in which case the maximum time spent in saving states of the system in the worst case is $f_j(r + t_j)$, or of inserting a rollback point ($x_{ij}(r)=1$) in which case the minimum time spent in saving states of the system in the worst case is $S_{ij} + f_j(L_{ij} + t_j)$. The optimal decision $x_{ij}(r)$ and the time spent in saving states of the system in the worst case after task $i$ and if task $j$ is processed next are clearly given by (1) and (2).

Since $f_j(r) = \min \{g_{ij}(r)\}$ it follows that $f_j(r)$ is the minimum time spent in saving states of the system after task $i$ in the worst possible case. This completes the proof of the theorem. Two examples, Figs. 5 and 6, have been worked out.

Minimal Expected Save-Time Design

Let $p_{ij}$ be the probability that task $j$ is followed by task $i$. Let $t_i, L_{ij}$, and $S_{ij}$ be expected rather than maximum values. Let us redefine $f_i(r)$ as

$$f_i(r) = \sum_j p_{ij} g_{ij}(r). \quad (3b)$$

The algorithm used in the worst-case design will determine the optimal decision rules for the minimal expected time design with (3a) replaced by (3b). The proof that the algorithm yields the optimal decision rules is similar to the proof of Theorem 1 and is not presented here.

Computation

The amount of computation is proportional to the sum of the number of vertices and edges in the graph. The computation of $f_i(r)$ and $g_{ij}(r)$ are straightforward, since all functions are of the form: $K_p$ for $q_p < r \leq q_{p+1}$, $p = 1, 2, \ldots T \ldots$, where the $K_p$ are constants. If all the data $t_i, L_{ij}, S_{ij}, M$, in the problem are integers, then clearly $q_p, p = 1, \ldots, T$ are also integers. Hence the maximum number of discontinuities $T$, in the functions $f_i(r)$ and $g_{ij}(r)$, cannot exceed $M$.

The computation is most efficiently carried out by means of lists. The list structures and list processing techniques used are described later.
SOLUTION TO EXAMPLE 1, FIG. 5

Initialization (0th Step): There is only one exit vertex (i.e., a vertex without successors), viz. vertex 7. Define

\[
f_1(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 25 \\
\infty, & \text{for } 25 < r.
\end{cases}
\]

Label vertex 7 (with a check \(\checkmark\)) to show that the function for vertex 7 has been determined.

1st Step: At this stage we note that only vertex 7 has been labeled. We note that vertices 5 and 6 have all their successors labeled. Compute \(g_0(r), x_0(r), \) and \(B_{07}\) from (1) and (2).

\[
g_{07}(r) = S_{07} + f_1(L_{07} + t_1), \quad \text{if } r + t_1 > M
\]

\[
\min \{f_1(r + t_1), S_{07} + f_1(L_{07} + t_1)\}, \quad \text{if } r + t_1 \leq M
\]

\[
S_{07} + f_1(L_{07} + t_1) = 2 + f_1(2 + 10) = 2 + f_1(12) = 2
\]

\[
f_1(r + t_1) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
\infty, & \text{for } 15 < r
\end{cases}
\]

hence

\[
g_{07}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
2, & \text{for } 15 < r \leq 25.
\end{cases}
\]

Compute \(x_0(r)\) from

\[
x_{07}(r) = \begin{cases} 
0, & \text{if } g_{07}(r) = f_1(r + t_1) \\
1, & \text{otherwise}
\end{cases}
\]

hence

\[
x_{07}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
1, & \text{for } 15 < r \leq 25
\end{cases}
\]

and

\[B_{07} = 15.\]

We next compute \(f_0(r)\) from \(f_0(r) = \max \) over all edges \((5, j)\) of \(\{g_{07}(r)\}\). Since there is only one edge \((5, 7)\) leaving vertex 5, we have

\[
f_0(r) = g_{07}(r), \quad \text{for } r \leq 25
\]

hence

\[
f_0(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
2, & \text{for } 15 < r \leq 25 \\
\infty, & \text{for } 25 < r.
\end{cases}
\]

Label vertex 6 to show that \(f_0(r)\) has been determined.

2nd Step: Now we compute \(g_0(r), x_0(r), B_{06}\) from (1) and (2), since vertex 5 has all of its successors labeled.

\[
g_{06}(r) = S_{06} + f_0(L_{06} + t_4), \quad \text{if } r + t_4 > M
\]

\[
\min \{f_0(r + t_4), S_{05} + f_1(L_{05} + t_4)\}, \quad \text{if } r + t_4 \leq M
\]

\[
S_{06} + f_0(L_{06} + t_4) = 2 + 0 + 2 + f_0(2 + 5) = 2 + f_0(7) = 2
\]

\[
f_0(r + t_4) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
2, & \text{for } 10 < r \leq 20 \\
\infty, & \text{for } 20 < r
\end{cases}
\]

hence

\[
g_{06}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
2, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 20 < r
\end{cases}
\]

Since \(g_{06}(r) = f_0(r + t_4)\) for \(r \leq 20\) we have

\[
x_{06}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 20 \\
1, & \text{for } 20 < r \leq 25
\end{cases}
\]

and

\[B_{06} = 20.\]

We now compute \(f_0(r)\) and since there is only one edge going out of node 3, namely edge \((3, 6)\), we get \(f_0(r)\) to be the same as \(g_{06}(r)\); hence

\[
f_0(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 15 \\
2, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 25 < r.
\end{cases}
\]

Label vertex 3 to show that \(f_0(r)\) has been determined.

3rd Step: At this point, vertices 5, 6, and 7 have been labeled. We note that vertices 3 and 4 have all their successors labeled. We compute \(g_{03}(r), x_{03}(r), \) and \(B_{03}\).

\[
S_{03} + f_0(L_{03} + t_4) = 2 + f_0(2 + 5) = 2 + f_0(7) = 2
\]

\[
f_0(r + t_4) = \begin{cases} 
0, & \text{for } 0 < r \leq 10 \\
2, & \text{for } 10 < r \leq 20 \\
\infty, & \text{for } 20 < r
\end{cases}
\]

hence

\[
g_{03}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 10 \\
2, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 25 < r
\end{cases}
\]

Since \(g_{03}(r) = f_0(r + t_4)\) for \(r \leq 20\) we have

\[
x_{03}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 20 \\
1, & \text{for } 20 < r \leq 25
\end{cases}
\]

and

\[B_{03} = 20.\]

We next compute \(f_0(r)\) from \(f_0(r) = \max \) over all edges \((5, j)\) of \(\{g_{03}(r)\}\). Since there is only one edge \((5, 7)\) leaving vertex 5, we have

\[
f_0(r) = g_{03}(r), \quad \text{for } r \leq 25
\]

hence

\[
f_0(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 10 \\
1, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 25 < r.
\end{cases}
\]

Label vertex 3 to show that \(f_0(r)\) has been determined.

4th Step: We note that vertex 4 has all of its successors labeled. Hence we similarly compute \(g_{04}(r), x_{04}(r), B_{04}, \) and \(f_4(r)\)

\[
g_{04}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 10 \\
1, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 25 < r
\end{cases}
\]

\[
x_{04}(r) = \begin{cases} 
0, & \text{for } 0 < r \leq 10 \\
1, & \text{for } 10 < r \leq 25 \\
\infty, & \text{for } 25 < r
\end{cases}
\]

\[B_{04} = 10.\]
Suppose that in a given run of the above program tasks $1, 2, 4, 6,$ and 7 are executed in sequence, and suppose all tasks take 3 units of time. Then no rollback points will be inserted. If on another run of the same program, the same tasks, 1, 2, 4, 6, and 7 are executed in sequence, and all tasks that take 5 units of time, a single rollback point will be inserted on edge $(4, 6)$; i.e., the system environment will be saved after task 4 is complete and before task 6 is initiated. If on yet another run of the same program, the same tasks are executed, and each task takes its maximum time, i.e., task 1 takes 10 units of time, task 2 takes 15 units of time etc. rollback points will be inserted between tasks 1 and 2, 2 and 4, 4 and 6, but not between 6 and 7. Hence, we see that real-time decisions are made as to which edges are to contain rollback points; these decisions are clearly a function of input data. However, even though the insertion of rollback points is dynamic, the implementation of the algorithm is very simple, and requires negligible overhead, once the breakpoints $B_{ij}$ are determined. Note that the breakpoints $B_{ij}$ themselves are not a function of input data.

**Solution to Example 2, Fig. 6**

**0th Step:** Vertex 1 is an exit vertex. Put

$$f_2(r) = \begin{cases} 0, & \text{for } 0 < r \leq 10 \\ \infty, & \text{for } 10 < r. \end{cases}$$

**1st Step:** Since vertex 1 has all of its successors labeled, we find $g_{12}(r)$ and $f_3(r)$.

$$f_3(r + t_4) = \begin{cases} 0, & \text{for } 0 < r \leq 1 \\ \infty, & \text{for } 1 < r. \end{cases}$$

$$S_{12} + f_3(L_{12} + t_4) = 2 + f_3(2 + 9) = 2 + f_3(11) = \infty$$

hence

$$g_{12}(r) = \begin{cases} 0, & \text{for } 0 < r \leq 1 \\ \infty, & \text{for } 1 < r. \end{cases}$$

Clearly $f_3(r) = g_{12}(r)$, since there is only one edge out of vertex 1.

Let the time $L_0$ taken to load the program initially be 1 unit. Then, since $f_3(L_0 + t_4) = f_3(1 + 2) = \infty$, there exists no feasible solution to the problem.

**The Code for the Algorithm**

A graphical example for computing $g_{ij}(r)$ is shown in Fig. 7, and of $f_i(r)$ is shown in Fig. 8.

Consider the problem shown in Fig. 9. The input data is of the form shown in Table 1. The first row, for instance, of Table 1 states that task 1 may be succeeded by task 2, and $L_{12} = 1$, $S_{12} = 1$. 

**5th Step:** At this stage vertices 3, 4, 5, 6, and 7 have been labeled. We note that vertex 2 has all of its successors labeled. We now compute $g_{23}(r), x_{23}(r), B_{23}$.

$$S_{23} + f_3(L_{23} + t_4) = 3 + f_3(3 + 5) = 3 + f_3(8) = 3$$

$$f_3(r + t_4) = \begin{cases} 0, & \text{for } 0 < r \leq 5 \\ 2, & \text{for } 5 < r \leq 20 \\ \infty, & \text{for } 20 < r \end{cases}$$

hence

$$g_{23}(r) = \begin{cases} 2, & \text{for } 0 < r \leq 5 \\ 3, & \text{for } 5 < r \leq 20 \\ 1, & \text{for } 20 < r \leq 25 \end{cases}$$

$$x_{23}(r) = \begin{cases} 0, & \text{for } 0 < r \leq 20 \\ 1, & \text{for } 20 < r \leq 25 \end{cases}$$

$$B_{23} = 20.$$ Similarly, we get

$$g_{24}(r) = \begin{cases} 1, & \text{for } 0 < r \leq 15 \\ 3, & \text{for } 15 < r \leq 25 \end{cases}$$

$$x_{24}(r) = \begin{cases} 0, & \text{for } 0 < r \leq 15 \\ 1, & \text{for } 15 < r \leq 25 \end{cases}$$

We label vertex 2 to show that $f_3(r)$ has been computed.

**6th Step:** At this stage vertices 2, 3, 4, 5, 6, and 7 have been labeled. We note that vertex 1 has all of its successors labeled; hence we compute $g_{12}(r), x_{12}(r), B_{12}$ and $g_{13}(r), x_{13}(r), B_{13}$.

$$g_{12}(r) = \begin{cases} 3, & \text{for } 0 < r \leq 10 \\ 5, & \text{for } 10 < r \leq 25 \end{cases}$$

$$x_{12}(r) = \begin{cases} 0, & \text{for } 0 < r \leq 10 \\ 1, & \text{for } 10 < r \leq 25 \end{cases}$$

$$B_{12} = 10$$

$$g_{13}(r) = \begin{cases} 1, & \text{for } 0 < r \leq 5 \\ 3, & \text{for } 5 < r \leq 25 \end{cases}$$

$$x_{13}(r) = \begin{cases} 0, & \text{for } 0 < r \leq 5 \\ 1, & \text{for } 5 < r \leq 25 \end{cases}$$

$$B_{13} = 5$$

hence

$$f_1(r) = \max \{ g_{13}(r), g_{12}(r) \}$$

$$\begin{cases} 3, & \text{for } 0 < r \leq 10 \\ 5, & \text{for } 10 < r \leq 25 \\ \infty, & \text{for } 25 < r. \end{cases}$$
The $g_{1j}(r)$ function is marked by diagonally hatched lines.

Fig. 7.

Combing $f_1(r)$ Given $g_{12}(r)$ and $g_{13}(r)$

Fig. 8.

Table I

<table>
<thead>
<tr>
<th>Task Number</th>
<th>Task Number</th>
<th>Load Time</th>
<th>Save Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The number of rows in the table = The number of edges in the graph.

Table II

<table>
<thead>
<tr>
<th>NODE LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Cell No. 1</td>
</tr>
<tr>
<td>Cell No. 2</td>
</tr>
<tr>
<td>Cell No. 3</td>
</tr>
<tr>
<td>Cell No. 4</td>
</tr>
<tr>
<td>Cell No. 5</td>
</tr>
</tbody>
</table>

Three linked lists are used for storing information. They are called the NODE, EDGE, and FUNCTION lists. Each cell in the NODE list has five fields (Table II). Cell (row) 3 of Table II states that vertex 3 has 1 (content of COUNT field) successor; the list of predecessors of vertex 3 starts in cell number 2 (content of TOP field) of the EDGE list; the list of successors starts at cell number 4 (content of BOTTOM field) of the EDGE list; the estimated maximum time of task 3 is 10 units; FLINK points to the cell in the FUNCTION list where the first element of the $f_1(r)$ function is stored.

Each cell in the EDGE list has seven fields (Table III). PREDECESSOR NODE and PREDECESSOR LINK fields are used to keep a linked list of the predecessors of a node. SUCCESSOR NODE and SUCCESSOR LINK fields are used to keep a linked list of the successors of a node. For instance the first cell in the list of predecessors of vertex 3 is cell number 2 of the EDGE
### TABLE III

<table>
<thead>
<tr>
<th>Edges</th>
<th>Successor</th>
<th>Predecessor</th>
<th>Function</th>
<th>Load Time</th>
<th>Save Time</th>
<th>Glink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

![Table III](image)

### TABLE IV

<table>
<thead>
<tr>
<th>Problem Number</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>220</td>
<td>429</td>
<td>0.099</td>
</tr>
<tr>
<td>2</td>
<td>225</td>
<td>419</td>
<td>0.087</td>
</tr>
<tr>
<td>3</td>
<td>176</td>
<td>318</td>
<td>0.068</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>397</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>230</td>
<td>469</td>
<td>0.104</td>
</tr>
<tr>
<td>6</td>
<td>233</td>
<td>446</td>
<td>0.095</td>
</tr>
<tr>
<td>7</td>
<td>182</td>
<td>361</td>
<td>0.076</td>
</tr>
<tr>
<td>8</td>
<td>188</td>
<td>370</td>
<td>0.085</td>
</tr>
<tr>
<td>9</td>
<td>222</td>
<td>434</td>
<td>0.095</td>
</tr>
<tr>
<td>10</td>
<td>244</td>
<td>447</td>
<td>0.098</td>
</tr>
<tr>
<td>11</td>
<td>223</td>
<td>429</td>
<td>0.098</td>
</tr>
<tr>
<td>12</td>
<td>201</td>
<td>388</td>
<td>0.087</td>
</tr>
<tr>
<td>13</td>
<td>192</td>
<td>382</td>
<td>0.086</td>
</tr>
<tr>
<td>14</td>
<td>226</td>
<td>458</td>
<td>0.096</td>
</tr>
<tr>
<td>15</td>
<td>212</td>
<td>410</td>
<td>0.093</td>
</tr>
<tr>
<td>16</td>
<td>217</td>
<td>424</td>
<td>0.096</td>
</tr>
<tr>
<td>17</td>
<td>200</td>
<td>380</td>
<td>0.077</td>
</tr>
<tr>
<td>18</td>
<td>216</td>
<td>417</td>
<td>0.091</td>
</tr>
<tr>
<td>19</td>
<td>192</td>
<td>377</td>
<td>0.086</td>
</tr>
<tr>
<td>20</td>
<td>230</td>
<td>434</td>
<td>0.098</td>
</tr>
<tr>
<td>21</td>
<td>220</td>
<td>456</td>
<td>0.102</td>
</tr>
<tr>
<td>22</td>
<td>206</td>
<td>400</td>
<td>0.078</td>
</tr>
<tr>
<td>23</td>
<td>161</td>
<td>302</td>
<td>0.067</td>
</tr>
<tr>
<td>24</td>
<td>229</td>
<td>449</td>
<td>0.105</td>
</tr>
<tr>
<td>25</td>
<td>217</td>
<td>431</td>
<td>0.103</td>
</tr>
<tr>
<td>26</td>
<td>215</td>
<td>421</td>
<td>0.088</td>
</tr>
<tr>
<td>27</td>
<td>192</td>
<td>361</td>
<td>0.084</td>
</tr>
<tr>
<td>28</td>
<td>254</td>
<td>480</td>
<td>0.110</td>
</tr>
<tr>
<td>29</td>
<td>175</td>
<td>350</td>
<td>0.083</td>
</tr>
<tr>
<td>30</td>
<td>179</td>
<td>325</td>
<td>0.073</td>
</tr>
</tbody>
</table>

![Table IV](image)

### Experimental Results

The algorithm has been coded in Fortran and run on a CDC 6600. Thirty problems were generated using the random number generator on the CDC 6600. Each problem had roughly 200 nodes and 400 edges. All problems were solved in less than 0.11 s. The computational results are shown in Table IV.

### Conclusions

The rollback problem has been described. Different models for the rollback problem have been compared and an optimal algorithm for one of the models has been presented. The list structures used in coding the algorithm have been discussed. Some experimental results obtained by running the code on a CDC 6600 have been presented.

The model has two possible drawbacks. Firstly, it is hard to accurately estimate the maximum execution times of only $N+M$ cells for the node and edge list. (A graph with 1000 nodes and 3000 edges needs only 4000 cells.) The size of the function list varies; the $f$ and $g$ functions are computed when required and the storage occupied by the $f$ and $g$ functions are returned to free storage when they are no longer required.

This method of storing the $f$ and $g$ functions is economical and allows for easy computation. For instance, evaluating $f(r)$ from $f(r) = \max \{g_{ij}(r), \ldots, g_{ij}(r)\}$ can be done readily by merely inspecting the points of discontinuity (i.e., the contents of the $X$ and $Y$ fields of the cells) of $g_{ij}(r), \ldots, g_{ij}(r)$.

The function list has three fields: $X$, $Y$, and XYLINK.

Note that a task graph with $M$ nodes and $N$ edges needs the predecessor node field of cell number 2 of the edge list is 2, since vertex 2 is a predecessor of vertex 3. The predecessor link field links the list of predecessors. The load- and save-time fields are self-explanatory, the load time of 3 and the save time of 3 in the second cell of the edge list refer to edge (2, 3). XYLINK points to a cell in the function list where the first element of the $g_{2}(r)$ function is stored.

The $f(\ )$ and $g(\ )$ functions are step functions. They are stored by storing the breakpoints. For instance, a step function and the method of storing it as a linked list are shown in Fig. 10. The $f$ and $g$ functions are stored as linked lists; each cell in the function list has three fields: $X$, $Y$, and XYLINK.
Detection of Multiple Faults in Combinational Logic Networks

IGAL KOHAVI, MEMBER, IEEE, AND ZVI KOHAVI, MEMBER, IEEE

Abstract—New techniques are presented for generating fault-detection experiments for combinational logic networks. Only single-output functions are considered. Test-covering and test-equivalence relations between networks are defined and these relations are shown to be instrumental in generating the experiments. The techniques presented provide minimal experiments for detecting multiple faults in two-level networks and provide nearly minimal experiments for most other networks.

Index Terms—Combinational logic, diagnosis, fault detection, multiple faults, test equivalence, testing.

INTRODUCTION

The simplest approach to determine whether a logical network is operating correctly is to apply to it every possible input combination and compare the resultant output with either the corresponding truth table or a faultless version of the same network. Such exhaustive tests [2], [3], however, are generally very long and for most networks are unnecessary, since it is usually possible to detect faults within a network by considerably shorter tests. Several methods have been developed [1], [4]–[9] for generating shorter and more efficient testing procedures. At present, however, it seems that an effective algorithm is still to be found.

It is convenient to distinguish between a fault-detection test, which refers to a single application of values to the input terminals, and a fault-detection experiment, which refers to a

tasks. Secondly, an accurate description of a program may require that the program graph have a very large number of nodes. The second drawback is initiated since the algorithm is efficient and does not require much storage or processing time to analyze large graphs.

As in many modeling problems these days, the major “cost” of using the model is the time required to obtain data and estimates rather than the time required to run the algorithm on a computer. Improvements should primarily be concerned with models using less data and fewer estimates.

ACKNOWLEDGMENT

The authors would like to thank A. Cowan for his programming help.

REFERENCES